Dynamic Monitoring of High-Performance Distributed Applications

Dan Gunter, Brian Tierney, Keith Jackson, Jason Lee, Martin Stoufer

Computing Sciences Directorate
Lawrence Berkeley National Laboratory
University of California, Berkeley, CA, 94720

Abstract

Developers and users of high-performance distributed systems often observe performance problems such as unexpectedly low throughput or high latency. Determining the source of the performance problems requires detailed end-to-end instrumentation of all components, including the applications, operating systems, hosts, and networks. However, one must be very careful to design the instrumentation to have extremely low overhead, and not affect the system being monitored. In this paper we present a very light-weight instrumentation system that can be dynamically activated to unobtrusively collect and aggregate detailed end-to-end monitoring information from distributed applications. We also show how emerging "Web Services" can be used to facilitate remote interaction with this system.

1.0 Introduction

Developers and users of high-performance distributed systems often see unexpected performance problems. It can be difficult to track down the cause of these performance problems because distributed system components interact in complex ways. Bottlenecks can occur in any of the components through which the data flows: the applications, the operating systems, the device drivers, the network interfaces, and/or in network hardware such as switches and routers.

In previous work we have shown that detailed application monitoring is vital for both performance analysis and application debugging [34][4][33]. In general we have found that performance analysis of distributed systems requires monitoring events before and after every I/O operation. This can generate huge amounts of monitoring data, and great care must be taken to deal with this data in an efficient and unobtrusive manner. In large cross-domain systems such as computational or data Grids, fine-grained mechanisms for dynamic control of the monitoring are also essential.

Consider the use-case of monitoring some of the High Energy Physics (HEP) Grid projects [25][3][12] in a Data Grid environment. These projects, which will handle hundreds of terabytes of data, require detailed instrumentation data to understand and optimize their data transfers. For example, the user of a Grid File Replication service [5][38] notices that generating new replicas is taking much longer than it did last week. The user has no idea why performance has changed. Is there a problem in the network, disk, end host, GridFTP server, GridFTP client, or some other Grid middleware such as the authentication or authorization system? Monitoring information is needed to pinpoint the bottleneck, and determine what changed to cause this bottleneck. Current performance must be analyzed, and compared against a baseline drawn from previously archived information. This performance analysis requires monitoring data for hosts (CPU, memory, disk), networks (bandwidth, latency, route), and the FTP client and server programs.

In the example above, the amount of monitoring data generated from a well-connected large FTP server could be considerable. Consider the case of an FTP server with a fast RAID disk array that is connected to a Gigabit-Ethernet network. The server is instrumented to log the start and end times for all network and disk read and writes, which are in blocks of 64 KBytes. If the server has 10 simultaneous clients, each transferring data at 10 MBytes per second, this will generate roughly 6250 events per second of monitoring data. Assuming each monitoring event is 50 Bytes, this equates to 313 KBytes/second, or 1.1 GBytes per hour, of monitoring data. Clearly, the instrumentation data needs to be very compact and efficient in order to generate and store this much monitoring data without perturbing the system.

In this paper we describe a new, very efficient, binary event format for NetLogger [34], our distributed application instrumentation toolkit. We also describe how

This paper published in the proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, July, 2002, Edinburgh, Scotland.

NetLogger instrumentation can be generated only when needed, incurring negligible overhead when unused. Then we look forward to the next logical step of interfacing this functionality with other Grid components by exposing it as a Grid "Web Service" [17].

2.0 Related Work

There are a number of systems that address application monitoring. *log4j*, part of the Apache Project [21], has produced a flexible library for Java application logging. However, the performance of *log4j* is far lower than is necessary for detailed monitoring, as is shown below in the results section. Another tool is Autopilot [26], which uses the flexible "self-describing data format" (SDDF), but also has some performance limitations.

There are several Grid event monitoring publication systems currently under development which are based on the Global Grid Forum's [19] "Grid Monitoring Architecture (GMA)" [32]. These include NWS [37], R-GMA [13], Code [27], and the Globus project's [14] Metacomputing Directory Service (MDS) [8]. However none of these systems are designed to handle streams of thousands of events per second in a non intrusive manner.

Other related work includes general purpose event handling systems, such as the CORBA event service [7], the JINI distributed event service [20], and the ECHO Event Service [11]. Of all of these, only ECHO is specifically concerned with performance. Our message format is similar in size and efficiency to PBIO [10], which is used in the ECHO system, but is simpler and more dynamic.

In addition, there are several groups exploring the use of Web Services for various types of event handling in a Grid environment, one of the first of which was Dennis Gannon's Group at the University of Indiana [29]. They are now developing a more general-purpose messaging system, called XEVENTS [39], which uses XML/SOAP messages. The performance of their system is limited by its use of XML messaging.

3.0 Monitoring Components

The system described in this paper has four main monitoring components: the *application instrumentation*, which produces the monitoring data; the *monitoring activation service*, which triggers instrumentation, collects the events, and sends them to the requested destinations; the *monitoring event receiver*, which consumes the monitoring data; and the *archive feeder*, which converts events to SQL records and loads them into an event archive. These components are illustrated in Figure 1. In this paper, we focus on the first two components.

In order for a monitoring system to have high end-to-end performance, none of the components can cause the pipeline to "block" while processing the data, as this could cause the application to block while trying to send the monitoring to the next component. Depending on the execution environment, potential bottlenecks exist on the network from the producer to consumer, and inserting events into the event archive database. To avoid blocking, the system must impedance-match slow data "sinks" with fast data "sources" by buffering data to disk at all bottleneck locations, as shown in Figure 1. This is similar to the approach taken by the Kangaroo system for copying data files [31].

Of course, if the *sustained* data rate exceeds the maximum speed of the slowest component the disk buffers will eventually fill and the pipeline will block. However, in many application debugging and tuning scenarios, high monitoring data rates come in *bursts*, for example for the duration of a file transfer or the run of a single set of parameters, between which there is only low-frequency "background" monitoring such as CPU or network probes.

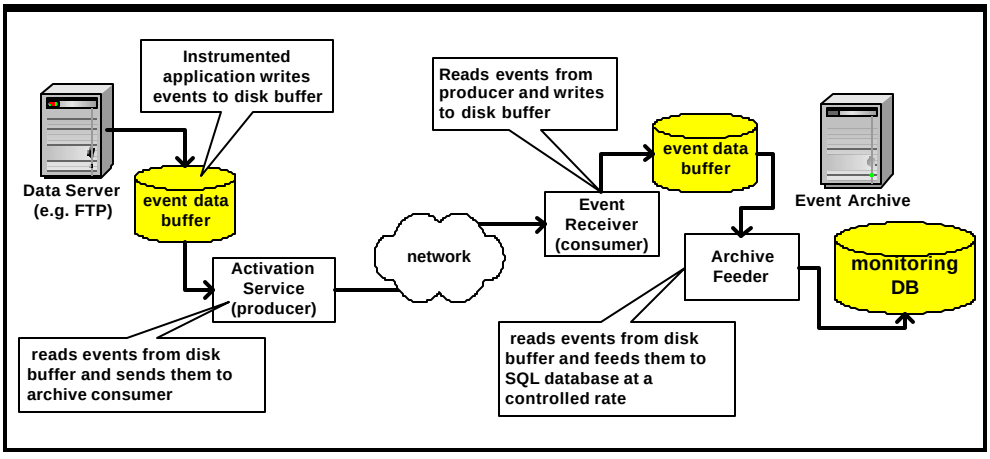

Figure 1: Monitoring Components

In this environment, the slower components will not block the pipeline, but only add some latency as the data waits in a buffer for processing.

4.0 NetLogger Toolkit

At Lawrence Berkeley National Lab we have developed the *NetLogger Toolkit* [34], which is designed to monitor, under actual operating conditions, the behavior of all the elements of the application-to-application communication path in order to determine exactly where time is spent within a complex system. Using NetLogger, distributed application components are modified to produce timestamped logs of "interesting" events at all the critical points of the distributed system. Events from each component are correlated, which allows one to characterize the performance of all aspects of the system and network in detail.

The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a set of tools for collecting and sorting log files, an event archive system, and a tool for visualization and analysis of the log files. In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then links the application with the NetLogger library. All the tools in the NetLogger Toolkit share a common log format, and assume the existence of accurate and synchronized system clocks. We have found that for this type of distributed systems analysis, clock synchronization of 1 millisecond is required, and that the NTP [24] tools that ship with most Unix systems (e.g.: *ntpd*) can easily provide this level of synchronization.

We have found exploratory, visual analysis of the log event data to be the most useful means of determining the causes of performance anomalies. The NetLogger Visualization tool, *nlv*, has been developed to provide a flexible and interactive graphical representation of system-level and application-level events.

Figure 2 shows sample *nlv* results, using a remote data copy application. The events being monitored are shown on the Y axis, and time is on the X axis. CPU and TCP CPU and TCP retransmit events are logged along with application events. Each lifeline represents one block of data, and one can easily see that occasionally a large amount of time is spent between Server_Send_Start and Client_Read_Start, which is the network data transfer time. From this plot it is easy to see that these delays are due to TCP retransmission errors on the network.

NetLogger's ability to correlate detailed application instrumentation data with host and network monitoring data has proven to be a very useful tuning and debugging technique for distributed application developers.

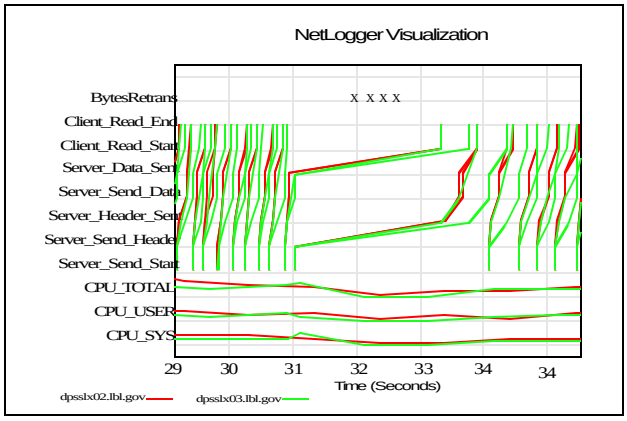

Figure 2: Sample NetLogger Results

4.1 NetLogger Binary Log Format

Previous versions of NetLogger used the IETF-proposed ULM format [1], a simple ASCII format based on name/value pairs. While easy to read and parse, this format imposed a great deal of unnecessary overhead. In order to improve efficiency, we have developed a new binary format that can still be used through the same API but that is several times faster and smaller, with performance comparable or better than binary message formats such as MPI [23], XDR [28], SDDF-Binary [26], and PBIO [10]. Our new NetLogger binary format is both highly efficient and inherently self-describing, thus, unlike the other formats, optimized for the dynamic message construction and parsing of application instrumentation.

The NetLogger binary message format is able to improve efficiency and simplify its design by restricting its expressiveness in a way appropriate to the monitoring domain. In the area of instrumentation and monitoring, the vast majority of communication can be naturally modeled with small time-stamped sets of name/value pairs. A few simple types -- floating-point numbers, integers, and strings -- account for almost all data values. For example, logging a transfer of a block of data would require a timestamp, host name, integer disk offset, and integer number of bytes. The NetLogger binary format restricts each message to contain less than 64KBytes, which reduces the need for dynamic memory allocation on both the sender and receiver. Only five simple types are allowed, thus simplifying the marshaling code: 32 and 64-bit integers, 32 and 64-bit floating point numbers, and strings.

In order to port the internal data representation across different architectures, NetLogger uses a methodology called "receiver-makes-right", in which the sender uses its native representation and the receiver converts if necessary. When both machines use the same internal representation, no conversion is necessary on either end,

but when conversion is necessary, the burden is always placed on the receiver. This is particularly appropriate for monitoring or application instrumentation, because minimizing perturbation at the sender, which is running the actual application code, is far more important than doing so at the receiver, which can even be a dedicated machine if necessary.

Monitoring data must be *self-describing*, so that it can easily be stored or processed by consumers of the monitoring data. The NetLogger binary format associates a descriptive "header" message with each type of "body" message. In the header, which is sent once per message type (per stream), are the name and data type of each data value in the message, and static values such as the program name or maximum data block size. Sending the constant fields only once means that the NetLogger binary format, unlike ULM or a generic binary encoding, does not incur a performance penalty for static descriptive fields in messages.

Finally, the importance of I/O buffering for high-volume logging is often overlooked. With relatively small messages, buffering can reduce the number of I/O calls by a factor of 10^3. Buffering increases communication latency, but even for "real-time" monitoring added latencies of a second or two are generally acceptable. The NetLogger library, by default, employs 128KB buffers that automatically flush when full or when one second has passed, thus limiting latency but minimizing the load on the system at high data rates.

4.2 NetLogger Message Generation Results

We gathered performance results for the binary message format in both C and Java. In both languages, we compared the binary format to NetLogger's ASCII ULM and XML formats. In C, we also ran the same test with Pablo's binary SDDF format and with PBIO. In Java we recreated the ULM log format with *log4j*, since log4j is a common solution for instrumentation of Java applications.

For each test we generated and logged timestamped events with the following information:

Timestamp=(microseconds); Host name="foo.lbl.gov"; Program name="MY_PROGRAM"; Event name="MY_EVENT"

Variables: "MY_INT"=(32-bit integer value); "MY_FLOAT"=(32-bit floating point value)

The timestamp and two variables change with every event, and the other fields are constant. Each test run generated 100,000 timestamped events (timestamps are obtained using the gettimeofday() system call for NetLogger and currentTimeMillis() for log4j). The average of five test runs was taken as the final result. For the Java testing, we preceded each set of five runs with a "warm-up" run to allow the Java Virtual Machine to optimize the bytecode. The test platform was an unloaded 750 MHz PIII host running Linux 2.4.16 and the Sun Java JVM version 1.3.1, logging to the Unix */dev/null* to avoid disk I/O issues. Results are shown in Table 1. For both C and Java, the highest throughput is clearly from binary NetLogger.

Table 1 Maximum speed results

Language	Test	Events/sec
C	NetLogger (binary)	615,000
C	PBIO (binary)	415,000
C	NetLogger (ascii)	175,000
C	SDDF (binary)	153,000
C	NetLogger (xml)	150,000
Java	NetLogger (binary)	122,000
Java	NetLogger (ascii)	97,000
Java	NetLogger (xml)	81,000
Java	log4j (ascii)	26,000

Note that, unlike NetLogger, *log4j* does not automatically timestamp and format the log entries. Our *log4j* test program spends a large portion of its time formatting the data string and building the output message. However, we feel that this reflects the normal usage pattern for *log4j*, and it is worth noting that our results are very close to those found on the *log4j* web pages [22], which used similar messages and an 800MHz PIII machine.

For the PBIO tests, we did not send the event name, program name, or host name with each message, as these remain constant for a given event type. We did generate and send a timestamp along with the int and float values. Although we did not do a full analysis, the fact that binary NetLogger is faster than PBIO is likely due to NetLogger's use of buffering.

The XML tests used the same low-level NetLogger library as the ULM tests, so the performance difference is entirely due to the increased verbosity of XML itself.

In summary; by borrowing ideas from systems such as SDDF and PBIO, and restricting their use to monitoring events with only five supported types, binary NetLogger is an extremely fast and unobtrusive solution for instrumenting applications.

5.0 Monitoring Activation

The Monitoring Activation Service is used to start and stop application-level monitoring in a NetLogger-instrumented application. Applications must use the

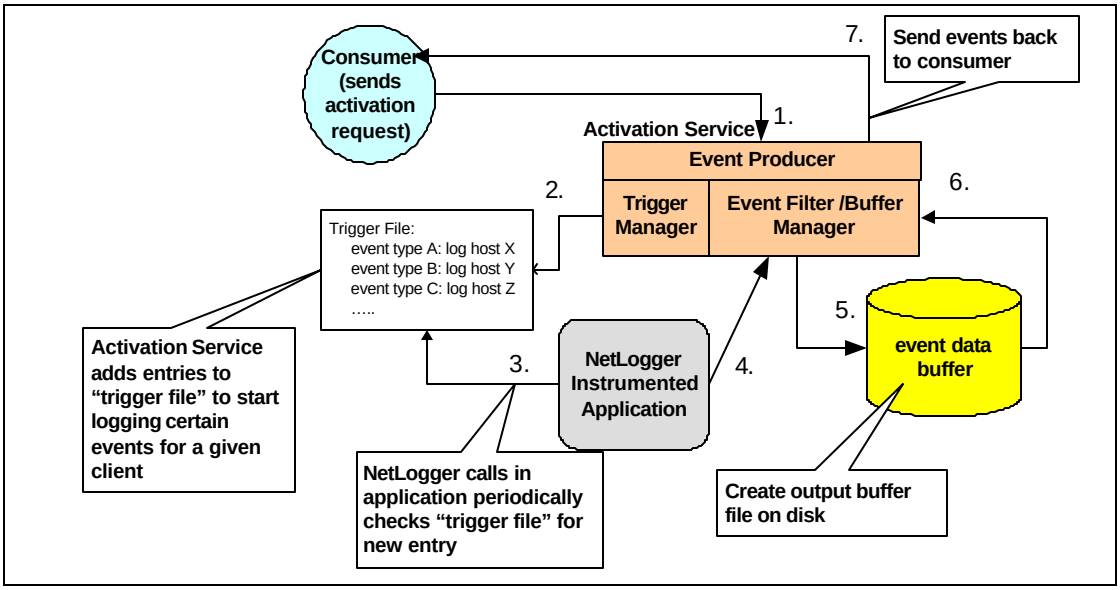

Figure 3: Client Request to the Monitoring Activation Service

NetLogger trigger API, a new addition to the NetLogger toolkit that uses an external file-based mechanism similar to the configuration files used by *log4j*. The trigger API allows long-running processes such as servers to change their logging behavior without command-line arguments, restarting, special signal handlers, or specialized control messages.

The method for implementing the trigger is a quite simple. At user-specified intervals (1 second by default), as part of a logging call, NetLogger will check the trigger file. If the file has changed, NetLogger will parse the file and determine if and where data written to this handle should be logged.

The components of the monitoring activation service are shown in Figure 3. This service waits for requests to send the data to some consumer (1), then creates a trigger file entry for a given event type (2). NetLogger calls in the application automatically check the trigger (3), and start sending NetLogger output to the activation service (4). The activation service reads the output and buffers it to disk (5). In parallel, it reads data from the disk buffer (6), and sends it to the consumer specified in the original request (7).

The activation service provides the ability to apply a client-specified filter to the monitoring data stream before buffering and forwarding. Initially, we have allowed clients to specify an event name, event name prefix, or simply all events. Next, we plan to allow the client to indicate interest in events with values above or below a threshold.

Performance Results

Standard binary NetLogger can generate 615,000 events per second. Binary NetLogger that is watching for trigger file updates can generate 583,000 events per second, or 5% less than when triggering is off. When the trigger file is modified it must be re-parsed. It takes 140μs to parse a large trigger file with 50 entries, which is quite tolerable at low update rates. The activation service's filtering introduces minimal overhead, roughly 1μs per event.

6.0 Use of Web Services

Web Services are an emerging set of standards for distributed internet computing, which build on XML-based protocols including SOAP [30] for transport, WSDL [6] for interface definition, and UDDI [36] for discovery, to create a framework that is independent of a particular language or programming model.

Web Services are fairly new, but the idea already has a great deal of momentum. The commercial and scientific computing communities as well as large companies -- including Microsoft, IBM, and Sun -- have committed to using Web Services. Many people in the W3C and other standards bodies are creating Web Service protocols. Soon we will see releases of open-source and commercial Web Service toolkits that can speed development of Grid services and distributed scientific applications.

We plan to deploy the Monitoring Activation Service as a Web Service. It will support the "producer" interface of the Grid Monitoring Architecture (GMA). Thus, consumers of monitoring data will be able use the same basic interface to get NetLogger application monitoring

data that they use to access other GMA systems such as the Globus MDS or the NWS. Also, the Monitoring Activation Web Service can use existing Web Services security work, such as SOAP over the Grid Security Infrastructure (GSI) [15], to authenticate requests for monitoring data.

The Grid Security Infrastructure is based on the Transport Layer Security (TLS) [9] protocol and X.509 Proxy Certificates [35]. TLS provides authentication and message integrity/confidentiality. GSI adds to TLS the ability to remotely delegate X.509 Proxy credentials.

Delegation is the main reason that the Activation Service needs to use GSI instead of the more common Secure Socket Layer (SSL) technology. The Activation Service will use the delegated credentials to activate monitoring on behalf of the requestor and to authenticate to third parties, such as an archive, that might receive the monitoring data.

By using standard commercial Web Service protocols, the Activation Service will be able to interoperate with commercial and Grid-based clients and services. For example, it will interoperate with services that use the Open Grid Services Architecture (OGSA) [16], another Web-services based architecture proposed by Argonne National Lab and IBM.

7.0 Case Study: GridFTP Server

The performance results given above for each component suggest that the entire system should be capable of handling the scenario given in Section 1. To test this assumption, we performed the following experiment.

For this experiment, we use the Globus GridFTP server [2], which has been instrumented using NetLogger to generate monitoring events before and after all I/O inside of the Globus I/O library [18]. A GridFTP server is installed on an 8 CPU Sun Solaris system (e4500), which is connected to a Gigabit Ethernet LAN. We did not have access to a fast enough RAID disk, so we are reading 200 MB files from /tmp, which on Solaris part of the virtual memory system, and has a read performance of 110 MB/sec.

Table 2 : NetLogger performance on the GridFTP server host

Test	Events/sec
C binary (1 thread)	384,000
C binary (5 threads)	1,520,000
C binary (8 threads)	2,440,000

The GridFTP server host used for this experiment is different from the Linux machine used to measure binary NetLogger throughput in Section4.2. Table 2 shows the NetLogger performance on this host, using the same messages as for the previous results. Because each thread uses a different CPU, the results correspond to the maximum speeds for 1, 5, and 8 or more simultaneous clients.

We first verified that the use of the NetLogger Activation Trigger did not add any significant overhead to the FTP server. We added NetLogger instrumentation to the server, and configured NetLogger to check for a trigger file once every second, but never added an entry to the trigger file. The results, shown in Table 3, show that adding NetLogger to GridFTP, but not activating it, adds only about 5% overhead, and had no effect on server throughput.

We then continuously ran tests with 1, 5, and 20 clients, each on a different hosts, all connected by Fast Ethernet or better networks to the server. We found that 20 clients could fully saturate the network link, but because the server was network-bound instead of CPU-bound, less than half of the available CPU cycles were used during the tests.

Table 3 : GridFTP server results

CPU time per transfer (seconds)			
Test	**mean**	**dev**	**delta**
1 client	4.2	4%	
1 client (un-activated Net-Logger)	4.4	4%	5%
1 client (NetLogger)	4.5	4%	7%
5 clients	5.5	5%	
5 clients (NetLogger)	5.8	5%	5%
20 clients	6.3	23%	
20 clients (NetLogger)	6.7	20%	6%
Aggregate throughput (Mb/s)			
Test	**mean**	**dev**	**delta**
1 client	73	1%	
1 client (un-activated Net-Logger)	73	<1%	<1%
1 client (NetLogger)	72	1%	1%
5 clients	361	1%	
5 clients (NetLogger)	363	1%	<1%
20 clients	441	4%	
20 clients (NetLogger)	409	5%	7%

The average event was 22 bytes. For 20 clients, the throughput of 409 Mbits/s generated an aggregate of roughly 3270 events per second. For one or five clients, the per-client event rate was roughly 580 events per second. For the NetLogger-instrumented runs, the trigger file was checked at 1 second intervals.

We calculated the CPU time per client by taking the sum of the 'user' and 'sys' time reported by the UNIX 'time' command for each execution of the FTP server, which gets started via *inetd* for each new client connection.

Results for these tests are shown in Table3. In this table, the "mean" is, for the CPU time, the 10% trimmed mean of CPU times for all clients, and the throughput is the sum of the 10% trimmed mean of each client's throughput. The "dev" is the percent deviation from the (non-trimmed) mean, and the "delta" is the percentage of increased CPU time or decreased throughput for NetLogger-instrumented runs as compared to non-NetLogger runs.

This table shows that adding NetLogger instrumentation with the trigger API has a small effect on CPU time and, except for the case of 20 clients, an even smaller effect on throughput. However, the increase in CPU time for all runs, and the decrease in throughput for the 20 client run, were higher than expected. We do not yet know the reason for this disparity.

Some of the additional overhead comes from the extra copy of the monitoring data which is buffered on local disk ('/tmp' in this case), and the separate process that copies it across the network to the final receiver. Triggering also accounts for a small increase in overhead, although we experimented with changing the trigger interval from 1 second to 5 seconds without an appreciable change in CPU overhead or throughput.

We believe that the results are on the whole encouraging. For a moderately loaded server with 360Mbits/s continuous throughput, NetLogger lowered throughput by less than one percent, and even when the network interface card was completely saturated, adding a binary NetLogger stream didn't increase per-client CPU time by more than 7 percent. Since most FTP servers are I/O limited, this small increase in CPU is negligible.

8.0 Conclusions

We have discussed the importance of detailed end-to-end monitoring data to analyze high-performance distributed applications such as GridFTP. We have also shown that, although a very high volume of monitoring information is generated by this process, NetLogger and the Monitoring Activation Service can dynamically provide real-time access to this information with minimal system perturbation.

9.0 Acknowledgments

This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. This is report no. LBNL-49698.

10.0 References

[1] Abela, J., T. Debeaupuis. *Universal Format for Logger Messages*. IETF Internet Draft, http://www.ietf.org/internet-drafts/draft-abela-ulm-05.txt

[2] Allcock B., Bester, J., Bresnahan, J., Chervenak, A., Foster, I., et.al. *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*. IEEE Mass Storage Conference, 2001.

[3] Avery, P. and Foster, I. The GriPhyN Project: Towards Petascale Virtual Data Grids. Technical Report GriPhyN-2001-15, 2001, http://www.griphyn.org/

[4] Bethel, W., B. Tierney, J. Lee, D. Gunter, S. Lau. *Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization*. Proceeding of the IEEE Supercomputing 2000 Conference, Nov. 2000.

[5] Cancio, G., S. Fisher, T. Folkes, F. Giacomini, W. Hoschek, D. Kelsey, B. Tierney. *The DataGrid Architecture*. http://grid-atf.web.cern.ch/grid-atf/doc/architecture-2001-07-02.pdf

[6] Christensen, E., F. Curbera, G. Meredith, S. Weerawarana. *Web Service Description Language (WSDL)* 1.1. W3C, Note 15, 2001, www.w3.org/TR/wsdl.

[7] CORBA. *Systems Management: Event Management Service.* X/Open Document Number: P437, http://www.opengroup.org/ onlinepubs/008356299/

[8] Czajkowski, K., S. Fitzgerald, I. Foster, C. Kesselman. *Grid Information Services for Distributed Resource Sharing*. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), August 2001.

[9] Dierks, T., and E. Rescorla. *The TLS Protocol Version 1.0*. Internet Draft draft-ietf-tls-rfc2246-bis-01.txt, March 2002.

[10] Eisenhauer G. and Lynn K. Daley. *Fast Heterogeneous Binary Data Interchange*. 9th Heterogeneous Computing Workshop (HCW 2000), May 2000.

[11] Eisenhauer, G., F. Bustamante, K. Schwan. *Event Services in High Performance Systems*. Cluster Computing: The Journal of Networks, Software Tools, and Applications, Vol 4, Num 3, July 2001, pp 243-252.

[12] European Data Grid Project http://www.eu-datagrid.org/

[13] Fisher, S. Relational Grid Monitoring Architecture Package, http://hepunx.rl.ac.uk/grid/wp3/releases.html

[14] Foster, I. and C. Kesselman. *Globus: A Toolkit-Based Grid Architecture*. In Foster, I. and Kesselman, C. eds. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999, 259-278.

[15] Foster, I., C. Kesselman, G. Tsudik, S. Tuecke. *A Security Architecture for Computational Grids*. Proc. 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92, 1998.

[16] Foster, I., C. Kesselman, J. Nick, S. Tuecke, *The Physiology of the Grid, An Open Grid Services Architecture for Distributed Systems Integration*, January, 2002, http://www.globus.org/research/papers/ogsa.pdf

[17] Graham, S., S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams, 2001.

[18] Globus IO: http://www.globus.org/v1.1/io/globus_io.html

[19] Global Grid Forum (GGF): http://www.globalgridforum.org/

[20] Jini Distributed Event Specification: http://www.sun.com/ jini/specs/

[21] log4j: http://jakarta.apache.org/log4j/docs/index.html

[22] log4j performance results: http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/performance/Logging.html

[23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. April 1994.

[24] Mills, D., *Simple Network Time Protocol (SNTP)*, RFC 1769, University of Delaware, March 1995. http://www.eecis.udel.edu/~ntp/

[25] Particle Physics Data Grid (PPDG): http://www.ppdg.net/

[26] Ribler, R., J. Vetter, H. Simitci, D. Reed. *Autopilot: Adaptive Control of Distributed Applications*. Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.

[27] Smith, W. *A Framework for Control and Observation in Distributed Environments*. NAS Technical Report Number: NAS-01-006, http://www.nas.nasa.gov/~wwsmith/

[28] Sun Microsystems. *XDR: External Data Representation Standard*. IETF RFC 1014, June 1987

[29] Slominski, A., M. Govindaraju, D. Gannon, R. Bramley. *An Extensible and Interoperable Event System Architecture Using SOAP*. http://www.extreme.indiana.edu/soap/

[30] Simple Object Access Protocol (SOAP) 1.1. W3C, Note 8, 2000. http://www.w3c.org/

[31] Thain, D., Jim Basney, Se-Chang Son, Miron Livny. *The Kangaroo Approach to Data Movement on the Grid*. Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing, San Francisco, California, August 2001

[32] Tierney, B., R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, M. Swany. *A Grid Monitoring Service Architecture*. Global Grid Forum White Paper. http://www-didc.lbl.gov/GridPerf/

[33] Tierney, B., D. Gunter, J. Becla, B. Jacobsen, D. Quarrie. *Using NetLogger for Distributed Systems Performance Analysis of the BaBar Data Analysis System*. Proceedings of Computers in High Energy Physics 2000 (CHEP 2000), Feb. 2000.

[34] Tierney, B., W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. Proceeding of IEEE High Performance Distributed Computing, July 1998, LBNL-42611. http://www-didc.lbl.gov/NetLogger/

[35] Tuecke, S., D. Engert, I. Foster, V. Welch, M. Thompson, L. Pearlman, and C. Kesselman. *Internet X.509 Public Key Infrastructure Proxy Certificate Profile*. Internet Draft draft-ietf-pkix-proxy-02.txt, February 2002.

[36] Universal Description, Discovery and Integration (UDDI): http://www.uddi.org.

[37] Wolski, R., N. Spring, J. Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. Future Generation Computing Systems, 1999. http://nsw.npaci.edu/

[38] Vazhkudai, S., S. Tuecke, I. Foster. *Replica selection in the Globus Data Grid*. International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001), IEEE Computer Society Press, 2001.

[39] XEVENTS project web page, http://www.extreme.indiana.edu/xgws/xevents/